CHAPTER

TWO

PROGRAMMING LANGUAGES

We assume the readers of this book are familiar with material covered in a typical undergraduate course on programming languages. Texts for such classes include Pratt [Pratt 75], Organick, Forsythe, and Plummer [Organick 75], and Ledgard and Marcotty [Ledgard 81]. However, not all readers have identical backgrounds. This chapter reviews two aspects of programming languages that are critical for the remainder of our discussion: the formal structure of languages (syntax and semantics) and the practical aspects of programming (pragmatics).

2-1 SYNTAX AND SEMANTICS

Programming languages are defined by their syntax and semantics. The *syntax* of a programming language specifies the strings of symbols that are legal programs. The *semantics* specifies the meaning of each syntactic structure—the action to be taken when that structure is encountered. Thus, the string of characters "3+5" might be a syntactically legal program segment. This string uses *numerals* "3" and "5" and a *plus sign*. Semantically, it might assert that the operation *addition* is to be performed on the *integers* 3 and 5.

Syntax

Perhaps the most useful broad classification of programming languages divides them into imperative and applicative languages. *Imperative languages* are statement- and sequence-oriented. Such languages associate names (variables) with the program state. A programmer can explicitly change the meaning of a name (and the program state) with assignment statements. Input and output are commands, executed like other commands. Repetition is shown by explicit repetitive statements (such as **for** and **while** loops). In general, imperative languages tend towards a complicated syntax, dependent on keywords and punctuation. These languages draw their inspiration from the architecture of the classical von Neumann computer and parallel the state orientation of automata like Turing machines. Fortran, Algol, Cobol, and Pascal are examples of imperative languages.

Applicative languages express programs through function application and binding. These languages associate names with values by function application and parameter binding. This binding provides just a name for the bound value, not the ability to change it. In an applicative system, inputs are the original arguments to the program/function, and the output is the function's result. Repetition is achieved by recursion. By and large, the syntax of applicative languages is more uniform than that of imperative languages. The classical applicative programming language, pure Lisp [McCarthy 60], is almost keyword-free. Lisp uses parentheses instead of keywords to show syntactic forms. Applicative languages are intellectual descendants of the lambda calculus. Pure Lisp and Backus's FP [Backus 78] are examples of applicative systems.*

The difference between imperative and applicative languages is primarily a difference of style. Modern languages frequently blend elements of each. Algol and Pascal permit functional subprograms and recursion; Lisp includes side effects and iteration. Nevertheless, the difference in appearance (though not necessarily substance) between the two forms is striking enough to merit noting. We will see contrasting examples of applicative and imperative styles in the remainder of the book.

Semantics

The syntax of a programming language describes which strings of symbols are legal programs. Programming languages would not be an interesting subject unless these strings could also effect actions. In general, the semantics of a programming language is the set of actions that programs in that language can perform. The semantics of a programming language associates particular actions (or functions) with particular syntactic structures.

* FP is actually a special variety of applicative system, a functional language. FP does not have variables or an abstraction operator (like lambda). It therefore does not need a renaming operation.

We can view the semantics of a particular programming language either with respect to computability theory, *formal semantics*, or with respect to the language's underlying conceptual model, *operational semantics*. Computability theory is concerned with the class of functions that can be computed by a given language. Many common languages, like Lisp and Algol, are computationally Turing-equivalent. The distributed languages we study are more computationally powerful than Turing machines, in that they can compute nonfunctional (multivalued) results. Computationally, such systems are equivalent to Turing machines that can also consult an unbounded random-number generator.

Every language designer has a model of the operational primitives that the language manipulates. In conventional computers, such primitives include arithmetic, logical, and input-output operations. Distributed language designers provide, in addition to these primitives, the primitives of the distributed domain. Typically, these primitives manipulate *processes* (automata), *messages* (communications between automata), and other abstractions. The semantics of any particular system defines what can be done with these objects—for example, how processes are created or how the order of message reception can be controlled.

A key theme in current programming language research is proving the correctness of programs. This involves specifying the meaning of each language construct, formalizing the problem statement, and showing that a given program solves the problem. Paralleling the division between imperative and applicative syntactic styles, there are two major themes in program semantics and verification: axiomatic and denotational. *Axiomatic* formalisms lend themselves to statement-oriented languages. These formalisms describe the state of the program execution by logical assertions before and after each statement. *Denotational* formalisms are oriented towards functional application. These formalisms build the meaning of a function application from the meanings of the function's arguments. Wand [Wand 80] includes a brief introduction to these themes. Thorough discussions of axiomatic semantics can be found in Alagic and Arbib [Alagic 78] and Gries [Gries 81], and of denotational semantics in Stoy [Stoy 77] and Gordon [Gordon 79].

2-2 PRAGMATIC STRUCTURES

Since programming languages are (by and large) formally equivalent, why does anyone bother to invent a new language? Language designers create new languages (and new programming structures) to aid in the practical aspects of programming — programming language *pragmatics*. For example, a programming language may provide both linked record structures and arrays. Pragmatically, arrays are used for direct access to fixed-size structures, while linked records are used for serial or logarithmic access to dynamic structures. However, nothing in the syntax or semantics of a language forces those choices. The choice among programming languages is usually one of pragmatics: the constructs of

the favored language simplify the creation of correct and efficient programs. Programming languages are themselves inherently pragmatic systems, designed to replace cumbersome assembly language programming.

In this section we consider the pragmatic aspects of three programming language concepts: data abstraction, indeterminacy, and concurrency.

Data Abstraction

Traditionally, programming languages have divided the world into programs and data. Programs are active; programs do things. Data is passive; data has things done to it. In a classical programming language, the ways of describing and using programs are distinct from the ways of describing and using data. There might be several ways of computing a particular function—for example, evaluating an arithmetic expression or looking up the value in a table. Classical programming languages make the choice of implementation obvious to any user.

Data abstraction provides an alternative to the fully visible programming style. Data abstraction merges the notions of program and data. This results in *objects*—program structures that can act both as program and as data. Objects hide the particular implementation of any given behavior from the user. Instead, a data abstraction system presents only an interface to an abstract object.

Typically, the data abstraction interface is just the set of operations that can be executed on an object. If the abstract object has associated storage, these operations often have side effects on that storage. For example, a data abstraction of finite sets would provide functions for testing whether an element is in a set, returning the size of a set, or generating the union of two sets. The data structure used to encode sets remains hidden. The set abstraction can be programmed in many different ways. For example, one can represent a set by a bit vector, a linked list, a hashed array, or even a program. Data abstraction permits the programmer (abstraction implementor) to choose the appropriate method for each individual set. The same abstraction can support several different implementations of sets simultaneously, with the differences invisible to the "user-level" programs. The implementor of a data abstraction can rely on the fact that routines that use an abstract data type have no access to the underlying representation that encodes that abstraction. One can safely modify the underlying implementation as long as it continues to satisfy the abstraction's specifications. Data abstraction is a cornerstone of many modern programming systems.

The earliest programming language implementation of data abstraction was the class mechanism of Simula 67 [Birtwistle 73]. Languages such as Smalltalk [Goldberg 83], CLU [Liskov 77], and Alphard [Shaw 81] have popularized the ideas of associating program with data and hiding implementation. Other programming language constructs that combine program and data include coroutines [Conway 63a], closures (a combination of environment and lambda-expression, see, for example, [Steele 78]), and thunks (a combination of environment and expression, [Ingerman 61]).

Indeterminacy

Most programming languages can describe only sequential, *determinate* programs. That is, given a particular input, the program has only a single possible execution path. *Guarded commands* is a programming language construct, introduced by Edsger Dijkstra [Dijkstra 75], that allows *indeterminacy*—any of several different execution paths may be possible for a given input.

Obviously, the most primitive way of stating, "do one of these statements," would be to have a disjunctive statement to that effect. Guarded commands take this idea one step further. To provide the programmer with greater control over which statements can be executed next, each of the candidate indeterminate actions is "guarded" by a boolean expression. That statement can be selected only if its boolean expression is true. Thus, a guarded command is both a syntax for indeterminacy and a variant of the conditional statement.

More specifically, we build a guard clause from a boolean guard B and a statement action S as

$$B \rightarrow S$$

We create a guarded command by joining several guard clauses with $\square$s

$$B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \ \square \ \cdots \ \square \ B_n \rightarrow S_n$$

To execute a guarded command, the system finds a boolean expression B_k whose value is "true" and executes the corresponding action S_k. The system simulates parallel guard evaluation.

Guarded commands whose guards are all false are treated differently in different languages. Some systems interpret this as an error, others as cause for exiting a surrounding loop, and still others as blocking the process until one of the guards becomes true. Of course, this last alternative is viable only for concurrent systems.

A program for computing the greatest common divisor of two numbers provides a simple illustration of guarded commands. We assume that a guarded command with false guards exits its surrounding loop. Euclid's algorithm for the greatest common divisor of two positive integers replaces the larger of the two with the difference between the larger and the smaller until the two are equal. As a program with a guarded command, this becomes

```
loop
      x > y   →   x := x − y
   □
      x < y   →   y := y − x
end loop
```

Variants of guarded commands are common in distributed programming languages. This is because guarded commands provide a concise way of reacting

to several possible different events without specifying a preferred or predicted order. Thus, a process expecting a message on one of several channels could have reception commands for each joined in a guarded command.

Concurrency

Guarded commands allow one of several alternatives to be executed, without specifying a preference among them. *Concurrent* statements allow several statements to appear to be executed "at the same time." Another name for concurrency is *parallelism*.

Syntactically, the simplest way to declare that several statements are concurrent is to have a structure that asserts, "execute these statements in parallel." A possible (imperative) syntax for concurrency is to wrap the concurrent statements between special delimiters, such as **parbegin** and **parend**:

parbegin
    statement$_1$;
    statement$_2$;
    statement$_3$;
        ⋮
    statement$_n$
parend

An alternative parallel syntax replaces the statement sequencer (;) with a different delimiter (||):

$$\text{statement}_1 \parallel \text{statement}_2 \parallel \text{statement}_3 \parallel \ldots \parallel \text{statement}_n$$

The corresponding applicative syntax would have a function whose arguments are evaluated in parallel. For the moment, we call the agent that executes an arm of a parallel statement a *process*. In Chapter 5, we present a more complete exposition of the notion of process.*

What does it mean to execute several statements concurrently? Each statement is composed of a sequence of indivisible primitives. For example, the expression

```
x := y + z
```

* Other ways of indicating parallelism include fork and join [Conway 63b] and systems based on the explicit declaration of processes. One problem with explicit parallel statements is the limitation of the degree of parallelism to the structure of the program text. A second problem is the lack of an explicit joining point for concurrent activities. Fork explicitly creates parallel processes; join uses counters to determine when a set of parallel activities has completed. Most of the languages for distributed computing discussed in Parts 2, 3, and 4 are based on explicit process declaration.

might be composed of the primitive steps

> Load the value of y into local register reg.
> Add the value of z to reg, keeping the result in reg.
> Store the contents of reg in x.

or, more symbolically,

```
reg := y;
reg := reg + z;
x   := reg
```

One way of viewing the semantics of concurrent statements is to see them as requiring the execution of some permutation of the indivisible primitives of their component statements, restricted only to keeping the primitives of a component in their original order. For example, if statement S is composed of indivisible primitives s_1, s_2, and s_3, and statement T is composed of t_1 and t_2, then the statement

parbegin
S;
T
parend

could be executed as any of the permutations

s_1, s_2, s_3, t_1, t_2
s_1, s_2, t_1, s_3, t_2
s_1, s_2, t_1, t_2, s_3
s_1, t_1, s_2, s_3, t_2
s_1, t_1, s_2, t_2, s_3
s_1, t_1, t_2, s_2, s_3
t_1, s_1, s_2, s_3, t_2
t_1, s_1, s_2, t_2, s_3
t_1, s_1, t_2, s_2, s_3
t_1, t_2, s_1, s_2, s_3

However, s_2 cannot be executed before s_1. The permutation mechanism hints at a critical postulate for concurrency: We cannot make any assumptions about the relative speeds of concurrent processes. One concurrent process may be arbitrarily (though not infinitely) quicker than another. Worse yet, the processes may be synchronized in lock step, so algorithms cannot be based on the possible occurrence of an irregular ordering.

Treating the semantics of concurrency as a permutation of events is an assumption about computational metaphysics. It asserts that events do not happen

simultaneously (or, similarly, that actions can be divided into atomic primitives). A world that allowed simultaneity would need to develop a semantics of simultaneous actions. In real computers, hardware arbiters minimize the probability of simultaneous events. Nevertheless, such arbiters can never completely exclude the possibility of failure from virtually simultaneous occurrences.

PROBLEMS

2-1 Show how conventional conditional statements could be replaced by guarded commands.

2-2 If statement M is composed of m primitive steps and statement N is composed of n primitive steps, how many different ways can the concurrent statement M||N be executed?

REFERENCES

[**Alagic 78**] Alagic, S., and M. A. Arbib, *The Design of Well-Structured and Correct Programs*, Springer-Verlag, New York (1978). This book presents the idea of integrating top-down development and correctness in the program development process.

[**Backus 78**] Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM*, vol. 21, no. 8 (August 1978), pp. 613–641. In this paper, the creator of Fortran describes a style of programs based on functional combination. This was Backus's Turing award lecture.

[**Birtwistle 73**] Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula Begin*, Auerbach, Philadelphia (1973). This book is an excellent introduction to Simula. Simula was originally designed as an extension of Algol 60 for systems simulation. Simula is important because its class mechanism is the intellectual ancestor of the current work on abstract data types and object-oriented programming.

[**Conway 63a**] Conway, M. E., "Design of a Separable Transition-Diagram Compiler," *CACM*, vol. 6, no. 7 (July 1963), pp. 396–408. This was the first article to describe coroutines.

[**Conway 63b**] Conway, M. E., "A Multiprocessor System Design," *Proceedings AFIPS 1963 Fall Joint Computer Conference*, AFIPS Conference Proceedings vol. 27, Spartan Books, New York (1963), pp. 139–146. Conway presents the fork and join primitive.

[**Dijkstra 75**] Dijkstra, E. W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," *CACM*, vol. 18, no. 8 (August 1975), pp. 453–457. In this paper Dijkstra introduced guarded commands and provided a formal definition of their meaning. Dijkstra allowed guarded clauses in two different contexts, bracketed by **if**, **fi** pairs to indicate a single test and by **do**, **od** pairs for a loop. The **if** construct would signal an error if it encountered all false guards, while the **do** construct interpreted this as the exit condition of the loop.

[**Goldberg 83**] Goldberg, A., and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, New York (1983). Smalltalk is a programming language based on the metaphor of communicating objects. This book is not only a comprehensive description of the Smalltalk language but also a discussion of implementation issues involved in building Smalltalk systems.

[**Gordon 79**] Gordon, M.J.C., *The Denotational Description of Programming Languages*, Springer-Verlag, New York (1979). This book is a good introduction to the use of denotational semantics for proving properties of programs.

[**Gries 81**] Gries, D., *The Science of Programming*, Springer-Verlag, New York (1981). Gries develops axiomatic correctness and ties it to program development. He argues that these mechanisms are the basis of scientific principles for program development.

[**Ingerman 61**] Ingerman, P., "Thunks," *CACM*, vol. 4, no. 1 (January 1961), pp. 55–58. A thunk is a pair composed of a code pointer and a static-chain (environment) pointer. Thunks are used in imperative languages such as Algol for both call-by-name and for passing labels and functions as arguments.

[**Ledgard 81**] Ledgard, H., and M. Marcotty, *The Programming Language Landscape*, Science Research Associates, Chicago (1981). Ledgard and Marcotty develop the principles of programming language design by studying a series of mini-languages that illustrate particular themes.

[**Liskov 77**] Liskov, B., A. Snyder, R. R. Atkinson, and J. C. Schaffert, "Abstraction Mechanisms in CLU," *CACM*, vol. 20, no. 8 (August 1977), pp. 564–576. CLU is a programming language developed around the theme of abstract data types. CLU was one of the first languages to separate the specification of a data type from its implementation.

[**McCarthy 60**] McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine," *CACM*, vol. 3, no. 4 (April 1960), pp. 184–195.

[**Organick 75**] Organick, E. I., A. I. Forsythe, and R. P. Plummer, *Programming Language Structures*, Academic Press, New York (1975). An introduction to programming languages that develops the run-time structure of systems based on the contour model.

[**Pratt 75**] Pratt, T. W., *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, New Jersey (1975). Pratt first develops the concepts of programming languages and then describes several major languages in terms of these concepts.

[**Shaw 81**] Shaw, M. (ed.), *Alphard: Form and Content*, Springer-Verlag, New York (1981). Shaw presents a collection of research papers and reports on Alphard. Alphard is a language based on abstract data types.

[**Steele 78**] Steele, G. L., Jr., and G. J. Sussman, "The Revised Report on SCHEME, a Dialect of LISP," Memo 452, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts (January 1978). Scheme is a language that features a lexically scoped Lisp with functions and continuations as first-class objects.

[**Stoy 77**] Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, M.I.T. Press, Cambridge, Massachusetts (1977).

[**Wand 80**] Wand, M., *Induction, Recursion, and Programming*, North Holland, New York (1980). This book is an elementary introduction to the "mathematics" of computer science. After developing the theory of sets and functions Wand presents both a denotational semantics for proving the correctness of Lisp-like programs and an axiomatic semantics for proving the correctness of imperative programs.